In Proceedings of VERIFY'06, The Third International Verification Workshop (Seattle, WA, August 15–16, 2006).

Establishing High Confidence in Code Implementations of Algorithms using Formal Verification of Pseudocode [⋆]

Myla Archer Elizabeth I. Leonard

Code 5546, Naval Research Laboratory
Washington, DC 20375
{archer,leonard}@itd.nrl.navy.mil

Abstract. Using a theorem prover to establish that a body of code correctly implements an algorithm is a task seldom undertaken because the effort required tends to be prohibitive. Direct reasoning about code in a particular programming language requires that some version of the language's semantics—e.g., axiomatic, operational, denotational—be used to determine the program correctness assertions to establish with the theorem prover. Any scheme for generating correctness assertions will be language-specific, and for languages with complex constructs, can be complex to implement and use. Direct reasoning about algorithms using a theorem prover can be not just difficult, but impossible, if the algorithms are (as is typical) specified using informal pseudocode. This paper outlines a scheme that falls short of full program verification yet provides high confidence in the correctness of an algorithm's implementation. The scheme uses *formal* pseudocode specifications, in a restricted language of `while` programs with (possibly recursive) procedure calls, to bridge from algorithm specifications to implementations in code. Each block of formal pseudocode is verified in the theorem prover PVS by translating it into a state machine model and proving a set of state invariants. High confidence in implementation correctness is achieved by combining verification of the pseudocode with traceability arguments relating the algorithm specification to the pseudocode representation and the pseudocode representation to the actual code.

1 Introduction

The use of a theorem prover to directly establish properties of code, particularly the functional correctness of the code, is rare because of the level of effort required. This level of effort can be reduced using appropriate tool support; however, the tool support must be based on a formal semantics for the programming language. Many existing tools rely on the use of axiomatic semantics, but tool support can also be based on other forms of definition such as operational or denotational. However, any tool for code verification must be language-specific, so a special tool or set of tools is required for each language. Further, correctly implementing tool support for reasoning about complex language constructs can be a difficult task.

An alternative to verifying code directly is to verify the algorithm it implements. Verifying correctness of the algorithm can provide evidence for the

[⋆] This research is funded by ONR.

correctness of the code provided there is a solid argument that the algorithm captures the code behavior. Unfortunately, direct reasoning about algorithms using a theorem prover can be impossible, if the algorithms are represented, as is typical, using informal pseudocode. Algorithms represented using informal pseudocode may, for example, use natural language descriptions of the results of operations or values to be returned. Though the intention is clear, one cannot reason directly about the algorithms with a mechanical theorem prover from informal representations. Thus, if this approach is taken, what is needed is a "formal" pseudocode representation for algorithms.

The above approach to establishing functional code correctness can also be turned around: start with an algorithm specification in (possibly informal) pseudocode. Then, translate the algorithms to formal pseudocode, verify the formal pseudocode, and finally, implement the pseudocode using the code constructs in the target programming language. In this paper, we describe how we are applying just such a process to create high confidence code. In cooperation with one colleague providing algorithm specifications and another colleague translating formal pseudocode into code in the programming languages C [13] and Ruby [17], we are providing and verifying the formal pseudocode tying specification to code, and establishing the correspondences between 1) formal pseudocode and specification and 2) between formal pseudocode and actual code.

Our verification of the formal pseudocode is based on a translation scheme that transforms a block of formal pseudocode annotated with assertions into a state machine representation and a set of invariant lemmas in PVS [15]. The state machine representation uses the automaton template of the TAME (Timed Automata Modeling Environment) [2, 3] interface to PVS; this permits the TAME support for proving state invariant properties of automata to be used in verifying the assertions in the formal pseudocode.

The contributions described in this paper include 1) a relatively inexpensive programming-language-independent method, based on verification of formal pseudocode, for providing high confidence that a program implements an informally specified algorithm; 2) an automatable translation scheme for transforming a set of related blocks of formal pseudocode with assertions into state machine specifications and candidate state invariants in PVS; and 3) tool support based on TAME for interactively proving the candidate invariants in PVS. The formal pseudocode serves as a bridge from algorithm to program. The TAME tool support allows the code annotations the candidate invariants to be weaker than would be required in a Floyd/Hoare-style verification. When they are weaker, the process of using TAME to create mechanized proofs of can lead to the discovery of needed "strengthening lemmas", i.e., auxiliary invariant lemmas that, if established, can be appealed to to complete the proofs of the candidate invariants. We have built a prototype translator, and have demonstrated the use of our translation and proof techniques, including discovery and proof of strengthening lemmas, on representative examples of asserted formal pseudocode.

This paper is organized as follows: Section 2 discusses the relationship of our work to the work of other authors. Section 3 begins by describing our language of formal pseudocode, and then describes how we represent formal pseudocode in TAME, and how the TAME proof support is used in proving properties of the TAME model. Section 4 provides details of how we use formal pseudocode as a bridge from algorithm specifications to code. Section 5 provides examples of algorithms, their formal pseudocode representation, TAME models of formal pseudocode, and TAME proofs of properties. Finally, Section 6 provides observations on our results so far, and Section 7 presents our conclusions.

2 Comparison to Related Work

The work of Boyer and Moore in [7] is similar to ours in that it describes a method for verifying programs annotated with Floyd style assertions. In contrast to our method, which verifies formal pseudocode, verification in [7] is applied directly to programs in a real programming language, namely, a large subset of FORTRAN. Rather than representing a FORTRAN program as a state machine, however, the method in [7] divides the flow graph of the program into a set of simple paths and then directly generates verification conditions for each path from annotations at the endpoints of the path. An example is given of implementation of an algorithm (see [6]) in (informal) pseudocode directly in FORTRAN, and of the generation of verification conditions formulated as suitable input for the Boyer-Moore theorem prover Nqthm. Thus, as in our method, traceability from algorithm to program is done offline. An intermediate formal pseudocode layer is not required because the tool support is designed specifically for FORTRAN.

Another effort similar to ours is the Praxis effort described in [5]. The Praxis method also documents traceability from algorithm (design) to program. The traceability information relies on similarity of structure between code and Z [16] specifications accompanied by explanatory text and diagrams; thus, as in our approach, it is semi-formal. Like the method of [7], the Praxis method requires that implementations be done in a specific programming language—in this case, SPARK Ada—as the support tools are built for SPARK Ada.

Many other efforts, such as those described in [8, 1, 10–12], specify the semantics of a programming language in a theorem prover, and then prove or derive Hoare style rules for reasoning about programs in the language. The theorem provers used include HOL [9], which was used in [8, 1, 10, 11], and PVS and Isabelle [14], which were both used in [12]. The language semantics is defined by various means: [8] uses a relational semantics of command expressions, [1] represents commands by predicate transformers, [10] and [11] use an operational semantics based on command syntax, and [12] uses a type theory based denotational semantics on command expressions. By contrast, we do not specify the semantics of our formal pseudocode language in PVS. Rather, the PVS representation of a program is as one or more state machines, and a translation tool external to the prover computes the state machine representations. At present, the correctness of our translation scheme has to rely on a pencil and paper proof.

In the methods of [7, 5, 8, 1, 10–12], correctness of programs is established by proving a set of verification conditions. The verification conditions for [5] derive from a standard semantics for the SPARK Ada subset of Ada and are proved using the SPARK Ada specific theorem prover SPADE. In [8] and [1], HOL tactics are used to generate the verification conditions. In [10, 11], a verification condition generator is defined in terms of HOL objects, and proved correct in

HOL. In [12], verification is done either directly from the denotational semantics of Java or indirectly using the derived Hoare style logic. Correctness is proved either by establishing invariant lemmas ("class invariants") or by proving verification conditions based on the Hoare style logic, and the user evidently must formulate the invariants and verification conditions by hand. In our approach, we must currently supply Floyd style inductive assertions by hand. We restrict our formal pseudocode language so that every program in our language is comprised of a set of blocks, each block serving as a procedure definition and ending in a `RETURN` statement. In each block, every line after an initial set of declarations is labeled, and each label has an associated assertion. In addition to assertions associated with its labels, each block is annotated with a precondition and a postcondition. The verification conditions of a block are lemmas stating that 1) the assertion at `L`i holds whenever the program label is `L`i and the precondition is satisfied, and 2) at the last label, the precondition implies the postcondition. A verification condition generator based on Hoare logic can potentially minimize the set of labels for which a user of our method must supply an assertion. The method of [7] follows such an approach.

Most of [7, 5, 8, 1, 10–12] note that the proofs of verification conditions require user guidance, and at least ([1]) investigates techniques for simplifying the user effort required in proofs. In [7], the proofs are (implicitly) done mechanically using Nqthm, which provides for inductive reasoning and can prove many assertions automatically. However, Nqthm sometimes needs user guidance in the form of auxiliary lemmas and proof hints. Our method simplifies the required user effort by incorporating the support for proofs of invariants provided in TAME. If it proves desirable, we may extend the TAME support by defining new PVS strategies specialized for state machine representations of programs.

Additional related efforts are discussed in the survey [4], which distinguishes them with respect to whether they use a *deep* embedding or a *shallow* embedding of the programming language in a theorem prover. In the conclusion of [4], the authors define the notion of a *hybrid* embedding in which there is an additional layer between the user and the theorem prover that performs the embedding. The additional layer is viewed as being an interface between a host logic and a guest logic. Our approach to reasoning about formal pseudocode programs in PVS seems closest to use of a hybrid embedding in which the programs are represented operationally as state machines.

3 Formal Pseudocode and Its Verification

To simplify our terminology, in the sequel we will refer to informal pseudocode as ip-code and formal pseudocode as fp-code. Like the language of the Sunrise Verification System [11], our fp-code language is an extension of the language of "`while` programs" that includes procedure calls. Our method of verifying fp-code is built on a translation scheme for representing asserted fp-code as a set of state machine models accompanied by a set of state invariants that capture the assertions. By representing the models using the automaton template of the PVS interface TAME[2, 3], we are able to use TAME's strategy support for verifying state invariants of automata to verify the assertions. We first describe programs in our fp-code language. Next, we describe how blocks of fp-code are represented as state machines in TAME, and how annotations are translated into invariants and invariant lemmas in TAME. Finally, we will describe how the TAME strategies can be used in proving the invariant lemmas.

3.1 Formal Pseudocode

An fp-code program consists of a set of basic blocks. Each basic block in fp-code is formulated as a procedure definition that follows certain conventions. Procedures will have two kinds of parameters: `VAR` and value parameters. The body of a procedure definition is a "`while` program" with (optional) constant and variable declarations and procedure calls. If present, the constant and variable declarations will precede the program statements in the procedure body. The body ends with its (unique) `RETURN` statement. By convention, every `VAR` parameter *param* of the procedure is echoed by a declaration of a constant *param*`_save` in the procedure body whose value is defined to be that of the `VAR` parameter. Since the constant declarations precede any program statements in the body, the constant *param*`_save` serves to capture the initial value of *param*. (To make the conventions described here and below less onerous for the fp-code writer, our translator includes a preprocessor that transforms more natural pseudocode into fp-code that follows the conventions.)

The variables (as well as any procedure parameters) may have any type for which a PVS representation has been specified for the fp-code-to-TAME translator; this set of "known types" currently includes

- the usual basic types: `int`, `nat`, `real`, `bool`, and enumerated types;
- structured types: arrays and records; and
- potentially, higher order types: sets and functions.

The possible program statements in an fp-code block will include assignment statements, procedure calls, `if` and `if_then_else` statements, and `while` statements. The expressions in `if`, `if_then_else`, and `while` statement tests and on the right hand side of assignment statements can use any operators associated with a known type. Expressions are not allowed to have side effects.

To facilitate translation into a TAME model, the body of an fp-code block is organized so that each line serves one of six purposes: 1) declaration of one or more variables, 2) test and jump, 3) target of a jump (e.g., `ENDIF`, `ENDWHILE`, `RETURN`), 4) assignment of a value to a variable, 5) call to a procedure, or 6) a `SKIP` statement. All non-declaration lines in the body of an fp-code block have associated labels; the first line after the declarations is a `SKIP` statement.

Each fp-code block defining a procedure *proc* will have two associated (predicate) functions that are considered part of the procedure specification:

- *proc*`_argsOK`, the block precondition, which takes as arguments the arguments to the procedure and returns a boolean, and
- *proc*`_post`, the block postcondition, which takes as arguments the arguments to the procedure plus all *param*`_save` constants, and returns a boolean.

Each recursive procedure will in addition have an associated metric function:

- *proc*`_metric`, which takes as arguments the arguments to the procedure and returns a natural number.[1]

[1] The metric function could actually return a value in any well-founded set, provided the elements of the set are of a "known type".

Finally, all fp-code blocks will have annotations associated with each line of fp-code following the declarations (or equivalently, with each label), where each annotation represents facts desired to be true about the program state just before that line is executed. To be appropriate as input for our translator, the function and predicate symbols used in the annotations must be "known" in the same sense as the types associated with variables: i.e., a PVS declaration of each function and predicate used must be present in an appropriate PVS theory.

3.2 Representing formal pseudocode in TAME.

Parameters and variables. For a block of fp-code to be translatable to a valid TAME specification, any parameters and local variables should have a known type. The translator turns the value parameters into constants in the TAME representation of `B`, declared at the beginning of the TAME representation. Next, the translator treats each `VAR` parameter *`param`* and each local variable as a state variable in the TAME representation. The translator then uses each special constant *`param`*`_save` to specify the initial value of the state variable *`param`* in the state predicate `start` in the TAME representation.

Labels and assertions. In asserted fp-code, the precondition and postcondition of a block and the assertions associated with its labels are expressed as formulas involving state variables and known functions and predicates on the data types of the variables. As mentioned in Section 2, the verification conditions for the block are a set of state invariant lemmas derived from the block's pre- and postconditions, and the lemmas include in their hypotheses a reference to the current program label. Thus, every state machine derived from a block includes `label` as a state variable. The type of the variable `label` is an enumerated type.

For each label, it is straightforward to translate the assertion $\mathtt{A(L}i\mathtt{)}$ associated with label $\mathtt{L}i$ into an assertion $\hat{\mathtt{A}}\mathtt{(L}i\mathtt{)(s)}$ about the current state `s`: the translator simply replaces every reference to a state variable `v` in $\mathtt{A(L}i\mathtt{)}$ by a reference to its value `v(s)` in `s`. This works also for the block's postcondition `O`. Because the precondition `I` of any block refers only to values of the procedure's value parameters or initial values of its `VAR` parameters, the assertion `I` involves only constants; no variable references need to be translated. For each assertion $\mathtt{A(L}i\mathtt{)}$, the translator defines an invariant $\mathtt{Inv\_L}i$ on the state `s` of the form:

$$\mathtt{I} \wedge \mathtt{AT(L}i\mathtt{)(s)} \Rightarrow \hat{\mathtt{A}}\mathtt{(L}i\mathtt{)(s)}$$

where $\mathtt{AT(L}i\mathtt{)(s)} \Leftrightarrow \mathtt{label(s)} = \mathtt{L}i$. For the last label $\mathtt{L}n$, the translator also defines the *block specification property* $\mathtt{Inv\_}proc$ (where *proc* is the block's procedure name):

$$\mathtt{I} \wedge \mathtt{AT(L}n\mathtt{)(s)} \Rightarrow \hat{\mathtt{O}}\mathtt{(s)},$$

where $\hat{\mathtt{O}}\mathtt{(s)}$ is the translation of the postcondition `O`. For each $\mathtt{L}i$ labeling a call to a procedure *P*, the translator adds a "call correctness" invariant $\mathtt{Inv\_L}i\mathtt{\_}P$.

Verification conditions. The verification conditions produced by the translator are the state invariant lemmas associated with the invariants $\mathtt{Inv\_L}i$ and $\mathtt{Inv\_}proc$ and any generated call correctness invariants $\mathtt{Inv\_L}i\mathtt{\_}P$. The current translation scheme derives these invariant lemmas from the code and an "assertions specification" that associates assertions with program labels. The lemmas have the form

```
FORALL(s:states): reachable(s) ⇒ Inv_Li(s)     (one for each label Li)
FORALL(s:states): reachable(s) ⇒ Inv_Li_P(s)       (if P called at Li)
FORALL(s:states): reachable(s) ⇒ Inv_proc(s)
```

Because it is necessary to verify all invariant lemmas associated with the labels in `while` loops simultaneously, the translator also generates invariant lemmas for each outermost `while` loop in the fp-code of the form

```
FORALL(s:states): reachable(s) ⇒ Inv_Li_1(s) ∧...∧ Inv_Li_k(s)
```

where the labels $\texttt{L}i_1$ through $\texttt{L}i_k$ label all the lines in the loop.

Defining the set of actions. There is just one action in the state machine representation of a block of fp-code: the action `step(new_s)`. Here, the parameter `new_s` can be any state. Because the effects of procedure call actions are only partly specified by a procedure's pre- and postcondition specification, these effects must be treated in the model as if they are nondeterministic (even though they are deterministic). The parameter `new_s` can be used to specify, in the precondition of the action `step(new_s)`, the known properties of the new state resulting from a procedure call at `label(s)`.

Defining the operational semantics of transitions. Every transition in the state machine representing an fp-code block transforms the current state `s` into a next state `s′` with appropriate updates to particular state variables. The state variable `label` is updated according to the flow of control through the block. To model the control flow effect of a line in the block representing an assignment statement or procedure call the current value `L`*i* of `label(s)` is updated to `L`*(i+1)*. For `if_then`, `if_then_else`, and `while` constructs, the fp-code format follows the restriction that there will be an `IF ... THEN` or a `WHILE ... DO` test on one line, and, as appropriate to the particular construct, a separate `ELSE` line, and separate lines for an `ENDIF` or `ENDWHILE`. The `WHILE ... DO`, `ELSE`, `ENDIF`, and `ENDWHILE` lines serve as jump targets. The separate `ENDIF` and `ENDWHILE` lines also provide labels associated with assertions that summarize the overall effect of an `if_then`, `if_then_else`, or `while`. When `label(s)` corresponds to a test line, the value of `label(s′)` will be the label of the appropriate target line; the appropriate choice is made based on the value of the test expression in `s`.

In addition to `label(s)`, other state variables must be updated on a transition in two cases: 1) when the fp-code line corresponding to `label(s)` is an assignment statement, and 2) when it is a procedure call. In the case of an assignment statement, the value of the state variable corresponding to the variable assigned to is updated according to the value in state `s` of the expression on the right hand side of the assignment statement.

The case of a procedure call is more complicated. The effect of a call to a procedure *P* in state `s` (other than to update `label(s)`) is to update the state variables passed as `VAR` parameters to their new values, as captured in the assertion *P*`_post`. Because it is often the case that *P*`_post` captures only an important property of the new values rather than their specific value—e.g., a procedure to sort list `lis(s)` results in `lis(s′)` being sorted—the effect of a call to a procedure P must be captured as if it were partially nondeterministic. As noted above, this can be done using the state parameter `new_s` of the transition action `step(new_s)`. The technique is to use the precondition of `step(new_s)` to restrict the values in `new_s` of the state variables corresponding `VAR` parameters of P to values that are acceptable after P executes. These values are then used to update the corresponding variables in state `s`. For a nonrecursive procedure P, the precondition is of the form *P*`_argsOK(<OK_args>) =>` *P*`_post(<post_args>)`, where `<OK_args>` are the parameters of P and `<post_args>` are the parameters of P with an extra argument *param*`_save` for each `VAR` parameter *param* of P. When the procedure P is recursive, the precondition is modified to condition the desired result on the hypothesis that the value of *P*`_metric` is reduced. For each procedure call to P at a label `L`*i*, the translator automatically generates `lemma_L`*i*`_`*P*, subject to proof, stating that *P*`_OK` holds when `label(s)` is `L`*i* and, if P is recursive, the value of *P*`_metric` in state `s` is reduced from its initial value.

3.3 Proving formal pseudocode verification conditions in TAME.

The first TAME step in proving any of the invariant lemmas described above is `(auto_induct)`, which sets up a proof by induction over the reachable states and discharges the trivial subgoals, which usually include the base case (when `s` is the start state). The induction case is always the case of a transition on action `step(new_s)`. The result of the transition depends on the label in the current state `prestate`, i.e., on `label(prestate)`. The next TAME step in the induction case is `(adt_cases label(prestate))`, which breaks down the induction case into cases for each label and discharges the trivial cases automatically.

When the invariant lemma being proved involves a single invariant, a property is being shown by induction to hold at a particular label `L`*i*. Since the `step(new_s)` action does not affect the truth of any assertion associated with label `L`*i* unless the action changes the current state `prestate` into a state with label `L`*i*, the cases for most of the possible labels for `prestate` are discharged as trivial, leaving only the cases for labels from which the state can "jump" to label `L`*i*. Because of the nature of `while` programs, there will be at most two nontrivial cases. For each nontrivial case, one then uses the TAME step `(apply_inv_lemma "L`*j*`")` where `L`*j* is `label(prestate)`. The case when one is simultaneously proving the invariants in a loop, the proof proceeds similarly, except that there will be many more nontrivial cases.

In examples we have done, it has been necessary to formulate some additional invariant lemmas besides those that the translator can automatically generate. Proving and using such auxiliary lemmas substitutes for direct strengthening of the original set of assertions prior to their proof.

4 Traceability: Bridging from Algorithm to Code

The fp-code is used as a bridge between the algorithm specifications, given in ip-code, and the actual code. The ip-code is converted by hand into fp-code, which is verified as described in the preceding section, and then used in the development of the actual code. This section describes how the ip-code is converted into fp-code and the relationship between the fp-code and the actual code.

4.1 Deriving formal pseudocode from informal pseudocode

The first step in deriving fp-code from ip-code is to organize the ip-code into blocks. (There may be more than one way to do this.) Every block of ip-code will be represented as a procedure definition in fp-code. If the ip-code is already organized as a set of procedures, then the blocks may just follow that organization. A large block of straight line code or the body of a long ip-code procedure may be split into multiple functional blocks based on code structure and variable usage. For each block, associated preconditions and postconditions need to be provided. In addition, when a block represents a recursive procedure, a metric function will need to be provided that can be used to measure progress in the recursion. Informal versions of these necessary assertions or functions may already be present in ip-code procedures. Informal loop invariants for loops may also be provided. When some necessary assertions and measures are not provided, they may need to be inferred by examining any assertions supplied with the ip-code and the ip-code itself.

Once the ip-code is organized into blocks, each block is translated into a basic fp-code block—i.e., a procedure definition. A combination of dataflow and parameter analysis of the ip-code is used to determine which of its variables to treat as value or `VAR` parameters to the new fp-code procedure and which to treat as constants or local variables. For example, if the ip-code block is a procedure, any parameters not modified (by an assignment or procedure call) in its body are treated as value parameters in the fp-code and any variables used as return values are treated as `VAR` parameters. If the ip-code block is not a procedure, any variables live at the end of the block are represented as `VAR` parameters in the fp-code procedure. The initial segment of variable declarations in the body of the fp-code procedure is used to represent those variables whose values are modified in the ip-code block but that are *local to the block*. The ip-code block may not contain declarations of all the variables used in the block. For each variable to be treated as a constant or a local variable, we add either a constant (uninterpreted) or variable declaration as appropriate.

The ip-code may also be missing the definitions of datatypes used in the algorithms and it may assume the existence of operators on those datatypes. It is necessary in the fp-code to provide fp-code representations of those datatypes and fp-code that implements the operators. Once formal versions—that is, fp-code representations—of the ip-code variables, types, and expressions have been decided on, it is possible to represent the (informal) versions of the preconditions and postconditions, metrics, and loop invariants (if given) as a formal part of the fp-code specification. If the ip-code metric is not expressible in terms of the procedure parameters, then a new parameter must be added to the fp-code to capture the metric. This may require the postcondition to be modified to take into account the new metric parameter.

Because the fp-code representation of a block needs to be fully asserted (i.e., assertions associated with every line of code), intermediate assertions need to be provided. In our work so far, we have done these by hand. As mentioned in Section 2, a mechanical verification condition generator could ease this process.

4.2 Relating formal pseudocode to actual code

The relationship of the fp-code to the actual code is established by inspection. Every construct in the fp-code language has a corresponding language construct in programming languages such as C and Ruby. Thus, there will be structural similarities between the fp-code and the actual code that make the correspondence more transparent. In addition, the labels of the fp-code can be mapped to line numbers in the actual code to document the correspondence. If the actual code uses constructs not included in the fp-code language (e.g., a FOR loop), then an offline argument will need to be made that any code written with those constructs is semantically equivalent to the corresponding fragments of fp-code.

5 Examples

5.1 TAME translation/verification of asserted formal pseudocode

To illustrate how our translation scheme transforms blocks of fp-code into TAME specifications, we will provide an fp-code specification of the well known algorithm `quicksort`, which is defined in terms of a recursive procedure (which we also call `quicksort`) and a procedure `partition`.

Figures 1 and 2 show fp-code specifications for `partition` and `quicksort`, respectively. The TAME state machine representation for `partition` is shown in Figure 3, and that for `quicksort` is shown in Figure 4. The fp-code for

```
PROC partition(VAR f:[int->nat], L:int, H:int, VAR P:int)
  f_save: [int->nat] = f;
  P_save: int = P;
  VAR lo, hi: int;
  VAR v: nat;

  L0:  SKIP;

  L1:  lo := L;

  L2:  hi := H;

  L3:  v := f(lo);

  L4:  WHILE (hi > lo) DO
  L5:    WHILE (f(hi) >= v & hi > lo) DO
  L6:      hi := hi - 1;
  L7:    ENDWHILE;
  L8:    IF (hi > lo) THEN
  L9:      f := f WITH [(lo) := f(hi), (hi) := v];
 L10:      WHILE (f(lo) <= v & hi > lo) DO
 L11:        lo := lo + 1;
 L12:      ENDWHILE;
 L13:      f := f WITH [(hi) := f(lo), (lo) := v];
 L14:    ENDIF;
 L15:  ENDWHILE;

 L16:  P := lo;

 L17:  RETURN;
ENDPROC;

partition_argsOK: L <= H
partition_post: permutation(f,f_save,L,H) & partitions(P,f,L,H)
```

Fig. 1. Formal pseudocode for `partition`.

```
PROC quicksort(VAR f:[int -> nat], L:int, H:int)
  f_save: [int -> nat] = f;
  VAR P: int;
  L0:  SKIP;
  L1:  IF (H > L) THEN
  L2:    EXECUTE(partition(VAR f, L, H, VAR P));
  L3:    EXECUTE(quicksort(VAR f, L, P - 1));
  L4:    EXECUTE(quicksort(VAR f, P + 1, H));
  L5:  ENDIF;
  L6:  RETURN;
ENDPROC;
quicksort_argsOK: L <= H
quicksort_post: sorted(f) & permutation(f,f_save)
quicksort_metric: H - L
```

Fig. 2. Formal pseudocode for `quicksort`.

`quicksort` is very compact, since it consists mainly of procedure calls to `partition` and to itself. As described in Section 3.2, the translator makes significant use of the definitions of the predicates `partition_argsOK` and `partition_post` and of the predicates `quicksort_argsOK`, and `quicksort_post`, and the metric `quicksort_metric` in formulating the preconditions in `enabled_specific` for the program steps corresponding to the procedure calls of `quicksort`.

5.2 Translation of an algorithm to formal pseudocode

To illustrate the translation of an informally specified algorithm into formal pseudocode, we present both pseudocode specifications for an algorithm to unify two literals that returns two values: 1) a boolean that indicates whether the literals are unifiable, and 2) if they are unifiable, a substitution list that if applied to the literals will make them equivalent. Figure 5 shows the ip-code specification for `Unify2Lits`. The corresponding fp-code specification is shown in Figure 6. Note that the fp-code has an additional parameter `n`, which is used in specifying the metric and that the postcondition for the fp-code also refers to this variable.

6 Discussion

On our translation technique. Our translation method is designed not to require an extensive symbol table. Thus, for example, we use `VAR` tags on `VAR` arguments to procedure calls as a hint to the translator. The translator must, however, coordinate translation of a set of related procedures in order to have access to the definitions of *`P`*`_argsOK`, *`P`*`_post`, and *`P`*`_metric` for each procedure *`P`* external to, but called in, the one currently being translated. Our translator relies only on a correct syntax; type correctness of the generated state machine definition is established using the PVS type checker.

Currently, assertions in asserted fp-code must be specified as properties associated with labels rather than as properties interleaved with lines of fp-code. As noted in Section 2, we may add a capability to generate the needed properties from a minimal set such as input, output, and loop invariant assertions.

```
partition_decls: THEORY
  BEGIN
    code_proof_lib: LIBRARY = "../code_proof_lib"
    . . .
    IMPORTING code_proof_lib@sorting_thy
    LABEL: TYPE =
      {L0,L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16,L17};
    L,H: int;
    f_save: [int->nat];
    P_save: int;
    partition_argsOK(L,H:int,f:[int->nat],P:int):bool = (L <= H);
    partition_post(L,H:int, f_in,f_out:[int->nat], P_in,P_out:int):bool =
      permutation(f_out,f_in,L,H) & partitions(P_out,f_out,L,H);
    const_facts: AXIOM TRUE;
    states: TYPE =
      [# label: LABEL, f: [int -> nat], P: int, v: nat, lo: int, hi: int #];
    f(i:int,s:states):nat = f(s)(i);
    AT(l:LABEL, s:states):bool = (label(s) = l);
    actions: DATATYPE BEGIN step(new_s:states): step? END actions;
    OKstate?(s:states): bool = TRUE;
    enabled_general(a:actions, s:states): bool = TRUE;
    enabled_specific(a:actions, s:states): bool =
      CASES a OF step(new_s): NOT(label(s)) = L17 ENDCASES;
    trans(a:actions, s:states): states =
      CASES a OF
        step(new_s):
          CASES label(s) OF
           L0: s WITH [label := L1],
           L1: s WITH [label := L2, lo := L],
           L2: s WITH [label := L3, hi := H],
           L3: s WITH [label := L4, v := f(lo(s),s)],
           L4: IF (hi(s) > lo(s)) THEN s WITH [label := L5]
               ELSE s WITH [label := L15] ENDIF,
           L5: IF (f(hi(s),s) >= v(s) & hi(s) > lo(s)) THEN s WITH [label := L6]
               ELSE s WITH [label := L7] ENDIF,
           L6: s WITH [label := L5, hi := hi(s) - 1],
           L7: s WITH [label := L8],
           L8: IF (hi(s) > lo(s)) THEN s WITH [label := L9]
               ELSE s WITH [label := L14] ENDIF,
           L9: s WITH [label := L10, f := f(s) WITH [(lo(s)) := f(hi(s),s),
                                                   (hi(s)) := v(s)]],
           L10: IF (f(lo(s),s) <= v(s) & hi(s) > lo(s)) THEN
                  s WITH [label := L11]
                ELSE s WITH [label := L12] ENDIF,
           L11: s WITH [label := L10, lo := lo(s) + 1],
           L12: s WITH [label := L13],
           L13: s WITH [label := L14, f := f(s) WITH [(hi(s)) := f(lo(s),s),
                                                    (lo(s)) := v(s)]],
           L14: s WITH [label := L4],
           L15: s WITH [label := L16],
           L16: s WITH [label := L17, P := lo(s)],
           L17: s
         ENDCASES
       ENDCASES;
    OKtrans?(a:actions, s:states): bool = TRUE;
    enabled(a:actions, s:states): bool =
      enabled_general(a,s) & enabled_specific(a,s) &
      OKstate?(trans(a,s)) & OKtrans?(a,s);
    start(s:states): bool = s = s WITH [label := L0, f := f_save, P := P_save];
    IMPORTING timed_auto_lib@machine[states, actions, enabled, trans, start]
END partition_decls
```

Fig. 3. TAME representation of `partition` fp-code.

```
quicksort_decls: THEORY
  BEGIN
    code_proof_lib: LIBRARY = "../code_proof_lib"
    . . .
    IMPORTING code_proof_lib@sorting_thy
    L, H : int;
    f_save : [int -> nat];
    quicksort_argsOK(f:[int->nat],L,H:int):bool = (L <= H);
    quicksort_post(L,H:int,f_out,f_in:[int->nat]):bool =
      sorted(f_out,L,H) & permutation(f_out,f_in,L,H) &
      eq_outside(L,H,f_out,f_in);
    quicksort_metric(L,H:int,f:[int->nat]):nat =
      IF quicksort_argsOK(f,L,H) THEN (H - L) ELSE 0 ENDIF;
    const_facts: AXIOM TRUE;
    states: TYPE = [# label: LABEL, f: [int -> nat], P: int #];
    actions: DATATYPE BEGIN step(new_s:states): step? END actions;
    f(i:int,s:states):nat = f(s)(i);
    AT(l:LABEL, s:states):bool = (label(s) = l);
    OKstate?(s:states): bool = TRUE;
    enabled_general(a:actions, s:states): bool = TRUE;
    enabled_specific(a:actions, s:states): bool =
      CASES a OF
        step(new_s):
          CASES label(s) OF
            L0: TRUE,
            L1: TRUE,
            L2: (L <= H)
                  =>  (partitions(P(new_s),f(new_s),L,H) &
                       permutation(f(new_s),f(s),L,H)),
            L3: (quicksort_metric(L,P(s)-1,f(s)) < quicksort_metric(L,H,f_save)
                  => (quicksort_argsOK(f(s),L,P(s)-1) =>
                      quicksort_post(L,P(s)-1,f(new_s),f(s))))
              & (NOT(quicksort_argsOK(f(s),L,P(s)-1)) => new_s = s),
            L4: (quicksort_metric(P(s)+1,H,f(s)) < quicksort_metric(L,H,f_save))
                  => (quicksort_argsOK(f(s),P(s)+1,H) =>
                      quicksort_post(P(s)+1,H,f(new_s),f(s)))
              & (NOT(quicksort_argsOK(f(s),P(s)+1,H)) => new_s = s),
            L5: TRUE,
            L6: FALSE
          ENDCASES
       ENDCASES;
    trans(a:actions, s:states): states =
      CASES a OF
        step(new_s):
          CASES label(s) OF
           L0: s WITH [label := L1],
           L1: IF (H > L) THEN s WITH [label := L2]
               ELSE s WITH [label := L5] ENDIF,
           L2: s WITH [label := L3, f := f(new_s), P := P(new_s)],
           L3: s WITH [label := L4, f := f(new_s)],
           L4: s WITH [label := L5, f := f(new_s)],
           L5: s WITH [label := L6],
           L6: s
         ENDCASES
       ENDCASES;
    OKtrans?(a:actions, s:states): bool = TRUE;
    enabled(a:actions, s:states): bool =
      enabled_general(a,s) & enabled_specific(a,s) &
      OKstate?(trans(a,s)) & OKtrans?(a,s);
    start(s:states): bool = s = s WITH [label := L0, f := f_save ];
    IMPORTING timed_auto_lib@machine[states, actions, enabled, trans, start]
END quicksort_decls
```

Fig. 4. TAME representation of `quicksort` fp-code.

```
(unifiable: bool, sigma: subst_list) Unify2Lits(l1,l2:literals):
if (NOT(sign(l1) = sign(l2)) or NOT(pred(l1) = pred(l2)))
   then unifiable:= false
   else mismatch := false
        i := 1
        while (i <= arity(l1) & not(mismatch))
           if (arg(i,l1) notequal arg(i,l2))
             then mismatch := true
             else i:= i+1
        if mismatch
          then u_result,s_result := UnifyTerms(arg(i,l1),arg(i,l2))
               if u_result = false
                 then  unifiable:= false
                 else u_result2, s_result2 := Unify2Lits(l1 s_result, l2 s_result)
                      if u_result2 = false
                        then unifiable:=false
                        else unifiable:= true
                             sigma := s_result s_result2
          else unifiable:= true
               sigma:= sigma_id
Unify2Lits_argsOK: l1, l2 are literals
Unify2Lits_post: ((NOT(EXISTS(s:subst_list):  substitution(l1,s) = substitution(l2,s)))
       <=> (unifiable = false))
   AND ((EXISTS(s:subst_list): substitution(l1,s) = substitution(l2,s))
       <=> ((unifiable = true) AND (substitution(l1,sigma) = substitution(l2,sigma))))
Unify2Lits_metric: each time the recursive call is made l1.arity -i is smaller
```

Fig. 5. Informal pseudocode for `Unify2Lits`.

```
PROC Unify2Lits(VAR unifiable:bool, VAR sigma:subst_list, n:nat, l1,l2:literal):
unifiable_save: bool = unifiable;
sigma_save: subst_list = sigma;
VAR u_result, u_result2: bool;
VAR s_result, s_result2: subst_list;
VAR l1_sigma, l2_sigma: literal;
VAR i: nat;
 L0: SKIP;
 L1: IF (NOT(l1.sign = l2.sign) OR NOT(l1.pred = l2.pred)) THEN
 L2:      unifiable:= FALSE;
 L3: ELSE mismatch := FALSE;
 L4       i := n;
 L5:      WHILE ((i <= l1.arity) AND NOT(mismatch)) DO
 L6:        IF NOT(l1.arg[i] = l2.arg[i]) THEN
 L7:             mismatch := TRUE;
 L8:        ELSE i:= i+1;
 L9:        ENDIF;
L10:      ENDWHILE;
L11:      IF mismatch THEN
L12:           UnifyTerms(VAR u_result, VAR s_result, l1.arg[i],l2.arg[i]);
L13:           IF u_result = FALSE THEN
L14:             unifiable:= FALSE;
L15:           ELSE IF (i < l1.arity) THEN
L16:                     EXECUTE(apply_sub(VAR l1_sigma,l1,s_result));
L17:                     EXECUTE(apply_sub(VAR l2_sigma,l2, s_result));
L18:                     EXECUTE(Unify2Lits(VAR u_result2, VAR s_result2, i+1,
                                 l1_sigma, l2_sigma));
L19:                     IF u_result2 = FALSE THEN
L20:                          unifiable:= FALSE;
L21:                     ELSE unifiable:= TRUE;
L22:                          EXECUTE(compose(VAR sigma, s_result, s_result2));
L23:                     ENDIF;
L24:                ELSE unifiable:= TRUE;
L25:                     sigma:= s_result;
L26;                ENDIF;
L27:           ENDIF;
L28:      ELSE unifiable:= TRUE;
L29:           sigma:= sigma_id;
L30:      ENDIF;
L31: ENDIF;
L32: RETURN;
ENDPROC;
Unify2Lits_argsOK: n <= l1.arity
Unify2Lits_post: ((NOT(EXISTS(s:subst_list):  substitution(l1,s) = substitution(l2,s)))
       <=> (unifiable = FALSE))
   AND ((EXISTS(s:subst_list): substitution(l1,s) = substitution(l2,s))
       <=> ((unifiable = TRUE) AND
           (FORALL(i:int): (n<=i AND i<=l1.arity)
               => (substitution(l1.arg[i],sigma) = substitution(l2.arg[i],sigma)))))
Unify2Lits_metric: l1.arity - n
```

Fig. 6. Formal pseudocode for `Unify2Lits`.

On our `fp-code` verification technique. So far, our verification technique can establish partial correctness. Although it includes proving invariants capturing the permissibility of making procedure calls, it does not guarantee that either procedure calls or `while` loops will terminate. Adding support for proving total correctness is a goal for the future.

Besides being based on proving invariants of a state machine representation of fp-code, our technique differs from standard techniques used with Floyd-Hoare style verification in other ways. For example, in place of "logical variables", e.g. to capture initial values, we use special new constants in the program. Also, we do not need to generate verification conditions as in Floyd or Hoare logic. Rather, the analogs of these verification conditions arise naturally as subgoals in state invariant proofs.

As with any verification technique for programs involving complex data types, to be complete, properties of these data types and their operators and predicates must be formulated in PVS theories and proved in PVS, so that they can be used in proofs of verification conditions. We have done this for a theory of sorting of functions of type `[int->nat]`.

Note that confidence in our technique will require proving that our fp-code-to-TAME translation scheme generates a TAME representation such that the properties proved of the TAME model also hold for the fp-code. Verifying the translator implementation would provide even higher confidence, though this is a goal for the more distant future.

7 Conclusions and Future Work

We have presented a verification scheme that relies on formal pseudocode as a bridge between algorithm specifications and implementations. We have also described a method of verifying fp-code using the TAME interface to PVS. This approach falls short of full program verification, but provides evidence of the correctness of the algorithm and its implementation.

`Unify2Lits` is part of a set of algorithms to perform unification and resolution. We plan to write fp-code specifications for all of these algorithms, verify them, and show the correspondence of the fp-code to C and Ruby code implementing the algorithms. This effort will include development of supporting PVS theories for expressions and substitutions, and enhancement of TAME's proof support with strategies specialized for algorithm verification.

Acknowledgements

We thank Vicky Weissman who provided the first version of `Unify2Lits` ip-code, and our colleague Tom Macklin who is translating fp-code into executable code. We also acknowledge Bill Harrison who is implementing a more robust version of the pseudocode-to-TAME translator.

References

1. Sten Agerholm. Mechanizing program verification in HOL. In *Proc. 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 208–222. IEEE Computer Society Press, 1992.

2. Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Math. and Artif. Intel.*, 29(1-4):139–181, 2000. Published Feb., 2001.
3. Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.
4. A. Azurat and I.S.W.B. Prasetya. A survey on embedding programming logics in a theorem prover. Tech. Rept. UU-CS-2002-007, Comp. Sci. Dept., Utrecht Univ.
5. Janet Barnes, Rod Chapman, Randy Johnson, James Widmaier, David Cooper, and Bill Everett. Engineering the tokeneer enclave protection software. In *Proceedings of the First IEEE International Symposium on Secure Software Engineering (ISSSE'06)*, March 2006.
6. R. S. Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
7. R. S. Boyer and J Strother Moore. A verification condition generator for FORTRAN. In R. S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, 1981.
8. M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.
9. M. J. C. Gordon and T.F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
10. P. V. Homeier and D. F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In *Higher Order Logic Theorem Proving and Its Appl.s*, Lect. Notes in Comp. Sci. vol. 859, pages 269–284. Springer, 1994.
11. Peter V. Homeier. A User's Guide to Proving Programs Correct with the Sunrise Verification System. `http://www.cis.upenn.edu/~hol/sunrise/guide.pdf`, 2005.
12. Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, Univ. of Nijmegen, The Netherlands, 2000.
13. B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
14. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.
15. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide, Vers. 2.4. Tech. R., Comp. Sci. Lab., SRI Intl., Menlo Park, CA, Nov., 2001.
16. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 1992.
17. D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: the Pragmatic Programmer's Guide, 2nd Ed.* The Pragmatic Bookshelf, Raleigh, NC, USA, 2005.